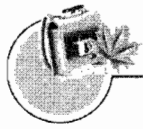

第7章

框　架

本章主要讨论的是有关驱动程序和文件系统的问题。

标题之所以这样取，是因为本章的侧重点不是去实现某一种硬件的驱动程序或支持某一个优秀的文件系统，而是从一个更抽象的角度去实现两个框架。任何驱动程序和文件系统都可以依据这种框架去实现它们的功能。这样，操作系统在与驱动程序和文件系统交互时，不再需要了解具体的驱动代码或文件系统结构，而只需要调用框架中的相关函数，从而实现了代码结构的抽象。同时，由于驱动程序五花八门、文件系统多种多样，旧的代码会被淘汰，新的功能会被添加，因此保持文件系统和驱动程序框架的相对稳定性，允许框架下的代码实时变化才是一个优秀的操作系统应具备的。这与前面我们反复强调的模块化的思想如出一辙。

当然，这些经验都是前人工作的结晶。因此，许多通用的操作系统都继承着这种结构，包括一些嵌入式操作系统以及几乎所有的非嵌入式操作系统。在本章中就让我们在学习其他操作系统优秀思想的同时，实现一下属于自己的框架。

7.1　驱动程序框架

在类 UNIX 操作系统中，根据传统的驱动程序框架，驱动程序被分成两大类：字符设备和块设备（因为网络设备稍微有些特殊，这里我们避开网络设备不谈）。

字符设备是指每次数据传输时将字符作为最基本单位的设备，如键盘、声卡等。它们通常不支持随机存取数据，因此对字符设备的处理非常直观简单，字符设备在实现时也大多不需要缓存，系统直接从设备读取/写入每一个字符。

而块设备则是与字符设备相对的另一个概念。它是指以批量方式进行数据传输的一类设备，如硬盘、CD-ROM 等。块设备通常支持随机存取和寻址并使用缓存。通常，操作系统会为块设备的输入或输出过程分配缓存，以存储临时读出或写入的数据。当缓存被填满时，会采取适当的操作把数据传走，而后系统清空缓存。这种针对块设备的缓存机制其实是广泛存在的，目的就是使访问速度较慢的块设备，能够适当地提高读写效率。

类 UNIX 系统将设备分成这样两大类，其实是对所有外设高度抽象化的结果。任何外设都可以被归类为按照字符读写的设备，或者被归类为按照块读写的设备。这样，在类 UNIX 操作系统中，内核只需要实现一个字符设备框架和一个块设备框架，就可以管理系统中所有设备了。

以上就是对类 UNIX 操作系统中设备管理的简单介绍。然而，想要在我们的操作系统中实现自己的驱动程序框架，却不能简简单单地照搬 UNIX 系统。

虽然类 UNIX 系统实现了设备的高度抽象，却在一定程度上牺牲了系统的整体效率。拿字符设备来说，类 UNIX 操作系统的驱动程序与文件系统共用同一种处理方法，也就是说，程序员只需要通过文件操作函数接口就可以直接操作设备了。这虽然统一了程序接口，但却是以降低程序运行效率为代价的。同时，为了迎合程序框架的通用结构，许多设备独有的特性不得不由驱动程序编写者去完成，因而提高了程序编写的复杂性。

就拿 Linux 系统下的摄像头为例，根据字符设备的分类原则，摄像头应该属于字符设备，于是在编写驱动时，就要根据字符设备框架的规定编写 file_operations 结构体，实现 open、write、ioctl 等函数（file_operations 是实现 Linux 字符设备的一个通用结构）。这样一来，针对摄像头这类设备的特有属性，如对颜色、格式等视频属性的处理，就不得不由驱动程序自己去完成，而每一个写驱动的人又几乎都要完成一些内容相似的东西，无论从效率上还是从稳定性上来看都不理想。

这就是传统设备驱动程序框架所带来的弊端。为了弥补这种不足，现在

这些系统通常都在原有传统设备框架的基础之上封装了各种各样针对不同设备的子框架，并让这种趋势渐成主流。

例如，Linux 中目前就有专门针对摄像头等媒体设备的 v4l 框架，有专门针对声卡设备的 alsa 框架等。这些驱动程序框架都是在原有的基于文件系统的框架基础上发展而来的，同时又更深入地解决了不同设备之间的差异问题。

那么在我们自己的操作系统中，不妨总结并发展类 UNIX 操作系统针对设备驱动的处理方法，一方面可以摒弃基于文件系统的设备框架，保证了效率，另一方面又为针对设备类型的框架提供了支持。因此，我们可以将设备分为若干类，每一类设备都有一套子框架对它做支持，同时保证以直接系统调用或类似的方式提供给用户，从而确保了内核结构的清晰和简单。

这便是我们设计驱动程序框架的思路。

7.1.1 基于存储设备的实例

下面我们以存储设备为例来学习驱动程序框架的应用。之所以选择存储设备，还有另外一个目的，那就是为后续章节的实践做好铺垫。

但在真正开始之前，我们还要解决一个非常棘手的问题，那就是设备。由于我们的操作系统一直都是在虚拟机中运行的，因此就不得不利用虚拟机虚拟出一块存储设备来。

出于多种原因，我们决定选用 RAM 盘作为例子来讲解设备框架。所谓 RAM 盘，其实就是将一块内存当成是诸如硬盘之类的非易失存储设备进行操作。从程序的角度来讲，RAM 盘是最简单的存储设备了，因为数据与存储器之间的交换可以通过原生的内存操作方法来实现，不像硬盘、存储卡等存储设备需要经过复杂的执行逻辑才可以将整块数据存入或读出存储器，这样就可以将重点放在程序结构上，而不是深陷到细节当中。

然而，框架的设计不同于写代码，需要对具体实例进行高度抽象。对于任意一种设备，我们都可以将针对它的操作方法抽象为五种类型：

- 设备的初始化操作
- 设备的释放操作

- 对设备的控制操作
- 数据写向设备的操作
- 数据从设备中读出的操作

存储设备作为设备的一个子类，自然也符合这五种基本的操作方法。就拿硬盘为例，系统开始运行时对硬盘控制器的配置就属于设备的初始化操作，向硬盘中复制数据就属于数据写向设备的操作，而更改硬盘的读写模式等，就可被归类为对设备的控制操作。

有的时候，有些设备可能不需要或没必要进行初始化和释放，对于这样的设备，设备的初始化和释放操作就可以被省略掉。同样，有些设备可能是只读的或只写的，比如一些简单的传感器，只需要从中读出数据即可，不需要进行数据的写入、初始化或控制。

在完成了对设备的抽象化过程后，我们需要做的就是将这种抽象实施到代码之中。此时我们已经将设备视为一种对象，并将以面向对象的方法去实践。

代码 7-1

```
#ifndef __STORAGE_H__
#define __STORAGE_H__

#define MAX_STORAGE_DEVICE (2)
#define RAMDISK 0

typedef unsigned int size_t;
struct storage_device{
    unsigned int start_pos;
    size_t sector_size;
    size_t storage_size;
int (*dout)(struct storage_device *sd,void *dest,\
                                    unsigned int bias,size_t size);
int (*din)(struct storage_device *sd,void *dest,\
                                    unsigned int bias,size_t size);
};

extern struct storage_device *storage[MAX_STORAGE_DEVICE];
extern int register_storage_device(struct storage_device *sd,\
unsigned int num);
```

```
#endif
```

在代码 7-1 中定义了 struct storage_device 结构体，该结构体专门用来描述一个通用的存储设备。

start_pos 成员用于描述这个存储设备的绝对位置，这个成员主要是为了解决对存储设备的分区问题而专门定义的；sector_size 成员用于描述存储设备的最小存储块，对存储设备的读写操作必须以这个值为基本单位，对于硬盘来说，这个值就是一个扇区的字节数。正因为存储设备是以块而不是以字节作为基本存储单位的，才使得存储设备具备了对大量数据进行存储的能力。storage_size 代表了这一个存储设备的总容量，函数指针 dout 和 din 分别指向了数据写入设备的函数和数据从设备中读出的函数。这两个函数的参数是相同的，都包含了写入的目的位置信息、数据源位置信息、待写入数据的大小，以及一个指向 struct storage_device 结构体的指针。其他诸如设备的初始化或控制的函数指针，这里省略没有列出。

举例来说，一个 NOR FLASH 设备一共有 4M 存储空间，如果在该设备上分出两个分区，分别为 1M 和 3M，那么就需要同时定义两个 struct storage_device 结构体：第一个结构体的 start_pos 应该为 0，storage_size 成员的值为 1M；第二个结构体的 start_pos 值为 1M，storage_size 的值就应该为 3M。由于 NOR FLASH 本身可以像内存一样，是不受一次读写数据大小的限制的，因此我们可以将这两个 struct storage_device 结构体的 sector_size 成员定义成 512 字节、1K 或者 2K 等常规数值。

代码 7-1 是使用 C 语言进行面向对象编程的典型方法，C 语言并非不适合进行以对象为基本思想的编程，它只是不够严谨和直接，真正起作用的是思想而不是语言本身。

既然已经有了描述抽象设备的基本类型，接下来我们就可以针对存储设备对这个类型进行实例化。

代码 7-2

```
#include "storage.h"

#define RAMDISK_SECTOR_SIZE    512
#define RAMDISK_SECTOR_MASK  (~(RAMDISK_SECTOR_SIZE-1))
#define RAMDISK_SECTOR_OFFSET  ((RAMDISK_SECTOR_SIZE-1))
```

```
extern void *memcpy(void * dest,const void *src,unsigned int
count);
int ramdisk_dout(struct storage_device *sd,void *dest,\
                 unsigned int addr,size_t size){
    memcpy(dest,(char *)(addr+sd->start_pos),size);
    return 0;
}

struct storage_device ramdisk_storage_device={
    .dout=ramdisk_dout,
    .sector_size=RAMDISK_SECTOR_SIZE,
    .storage_size=2*1024*1024,
    .start_pos=0x40800000,
};

int ramdisk_driver_init(void){
    int ret;
    remap_ll(0x30800000,0x40800000,2*1024*1024);
    ret=register_storage_device(&ramdisk_storage_device,RAMDISK);
    return ret;
}
```

代码 7-2 是一个超级简化版的 RAM 盘驱动程序。这段代码虽然简单，却足以说明驱动程序和框架之间是如何结合的。

写一个存储设备驱动程序，首先要做的就是准备一个 struct storage_device 结构体。在对该结构体进行必要的初始化之后，调用 register_storage_device 将它注册到系统之中。

在代码中定义的 struct storage_device 结构体名为 ramdisk_storage_device，该结构体的 start_pos、sector_size 和 storage_size 分别被定义成了 0x40800000、512 以及 2M。这表示这一存储设备的最小读写块为 512 字节，共有 2M 大小的空间，并且其起始的绝对地址是 0x40800000。请注意，这里的地址应该是内存虚拟地址，不要忘了我们的代码早已激活了 MMU 的功能。这也意味着在内存的物理地址某处，有 2M 大小的内存空间专门被当做 RAM 盘使用，并在使用这段空间时，需要将该物理地址映射到虚拟地址 0x40800000 处，这个动作可以依靠代码 7-3 中的 remap_ll 函数完成。

对于 ramdisk_storage_device 结构体来说，最关键的参数当属 dout 成员了。在代码 7-2 中，这一成员指向了函数 ramdisk_dout，其目的就是实现数据从 RAM 盘设备中的读取操作。根据代码 7-1 中的定义，该函数应有 4 个

参数，其中，dest 代表了数据将要被复制到的内存地址，size 表示复制数据的大小，可以是任意值，不必受限于存储设备的 sector_size 成员，而 addr 则代表了存储设备内部的偏移，它的取值范围为 0~storage_size。

我们要操作的硬件是一块 RAM 盘，因此实现数据的读取操作并不困难。函数 ramdisk_out 就是使用了 memcpy 来实现数据的读取。这个函数之前是定义在“print.c”文件里的，所以这里我们根本不需要实现，只需声明一下即可。

如果是其他存储设备，比如硬盘、NAND FLASH、SD 卡等呢？由于每一种存储设备都有自己的特点和操作方法，因此，dout 函数实现起来将会完全不同。但不管怎样，它的接口必须要符合 storage_device 中 dout 成员的定义。

代码中只实现了对存储设备的读的操作，写的操作与之类似，如果需要的话，读者朋友们可以自行完成。

当一切必要的成员都实现之后，就可以在 ramdisk_driver_init 函数中进行 RAM 盘设备的初始化了。

首先，我们要进行必要的内存映射，将物理地址某处的一块内存区映射到虚拟地址空间当中充当 RAM 盘进行使用。在前面几章的例子中我们都假设虚拟平台中有 8M 的内存空间，为了不改变前面的代码，保证代码的前后兼容性，本章中我们假设虚拟硬件中共有 10M 内存，其中前 8M 内存的划分方法与前面的代码相同，后 2M 内存则专门用来做 RAM 盘，这样一来，之前的程序就不需要修改了。

根据这样的假设，在我们的虚拟平台中 RAM 盘所处的物理内存地址就应该是 0x30800000~0x30a00000，紧接在系统原有的 8M 内存之后。此时我们就可以调用 remap_ll 函数进行地址映射了，该函数的实现方法如下：

代码 7-3

```
void remap_ll(unsigned int paddr,unsigned int vaddr,int size){
    unsigned int pte;
    unsigned int pte_addr;
    for(;size>0;size-=1<<20){
        pte=gen_ll_pte(paddr);
        pte_addr=gen_ll_pte_addr(L1_PTR_BASE_ADDR,vaddr);
        *(volatile unsigned int *)pte_addr=pte;
```

```
    }
}
```

如果读者已经完全理解了 MMU 的原理和操作方法，看这段代码就没有什么难度了。代码只是在循环里一次次地求出与物理地址以及要映射到的虚拟地址所对应的页表项和页表项的地址，再将页表项存入地址中去。

函数 ramdisk_driver_init 在完成了地址映射后，就可以调用 register_storage_device 函数将存储设备注册到系统中了。

代码 7-4

```
#include "storage.h"

struct storage_device *storage[MAX_STORAGE_DEVICE];

int register_storage_device(struct storage_device *sd,unsigned int
num){
    if(storage[num]){
        return -1;
    }else{
        storage[num]=sd;
    }
    return 0;
};
```

代码 7-4 只是示例性地定义了一个全局 storage_device 类型的数组，这个数组用来存储系统中所有的 storage_device 结构体，因为在一个硬件平台中很多时候并不会只有一种存储设备，当系统中既包含 NOR FLASH，又包含硬盘，同时还支持 RAM 盘时，将指向这些结构体类型的指针都保存到 storage 数组当中是一种非常简单的选择。而 register_storage_device 函数无非就是将某一个 storage_device 结构体赋值到 storage 数组相应的位置上。这样，我们就必须为每一种硬件都分配一个序号，就像代码 7-1 那样，RAMDISK 宏代表了 RAM 盘的硬件序号。注册的时候，首先检查 storage 数组的 RAMDISK 位置是不是 NULL，如果是，则表示有别的 RAM 盘驱动已经被注册了，如果没有注册，则将参数指针赋值给结构体数组的 RAMDISK 成员。

这样我们就利用了存储设备的驱动程序框架，编写出了一个简单的 RAM 盘驱动程序。作为本书的示例代码，我们尽量保证程序的精简和直观。

不过无论简单还是复杂，框架结构始终是统一的，本质的东西并没有什么变化。

7.1.2 运行存储设备实例

运行这段代码会稍微麻烦一些。首先我们必须修改“skyeye.conf”配置文件，新添加一块内存空间作为 RAM 盘。新的“skyeye.conf”内容如下：

代码 7-5

```
cpu:  arm920t
mach: s3c2410x

mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,
file=./leeos.bin,boot=yes
mem_bank: map=M, type=RW, addr=0x30800000, size=0x00200000,
file=./ram.img
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

这里我们定义了一个新的空间，从 0x30800000 开始，大小为 0x200000。在分配新空间的同时，又将一个名为“ram.img”的文件加载到这段空间内。“ram.img”文件的大小可以是 2M 或小于 2M，这其实是表示“ram.img”文件中的内容恰好代表了这段内存空间的内容。在实际的操作系统中，可以通过引导加载程序将相同的内容预先复制到该内存处去实现。

一个小问题随之而来，“ram.img”文件应该如何得到呢？其实这样的方法还是有很多的，无论在 Linux 环境还是 Cygwin 环境下，我们都可以借助于 dd 命令，其运行方法如下：

```
dd if=/dev/zero of=ram.img bs=1M count=2
```

这样，就恰好可以得到大小为 2M、内容全部为 0 的文件了。这条命令的意思是将设备文件 zero 中的内容复制出两块，每一块为 1M 大小，存储为“ram.img”文件。因为 zero 设备文件的内容全都是 0，因此“ram.img”的内容就全部是 0。

文件的内容可以是 0 或者是别的。我们使用“ram.img”文件的根本目的是在程序中读出从 0x30800000 开始，大小为 0x200000 的空间的任何一段内容，这其实就是文件本身的内容。然后将读出来的部分与文件做比较，以

此证明程序运行的正确性。

在完成上述工作后，将代码 7-4 的内容保存成名为“driver.c”的文件，再将代码 7-3 的内容添加到文件“mmu.c”的末尾，然后把代码 7-2 保存成文件，名为“ramdisk.c”，最后要保存代码 7-1 的内容，将文件命名为“storage.h”。

在所有的准备工作都完成后，为了验证这段代码的正确性，我们需要修改一下“boot.c”文件中的内容。

代码 7-6

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
//  timer_init();

    init_page_map();
    kmalloc_init();
    char buf[128];
    ramdisk_driver_init();
    storage[RAMDISK]->dout(storage[RAMDISK],buf,0,sizeof(buf));

    for(i=0;i<sizeof(buf);i++){
        printk("%d ",buf[i]);
    }
    printk("\n");

    while(1);
}
```

为了避免代码过长，我们删掉了以前的代码中各模块测试的相关部分，只保留了对块设备的测试代码。程序首先调用了 ramdisk_driver_init 函数进行 RAM 盘设备的初始化，然后通过 ramdisk_storage_device 结构体的 dout 成员将 RAM 盘内部 0 位置处的 128 个数据读出，保存到 buf 数组中，最后通过循环将数组中的数据读出。

修改“boot.c”文件的同时还要记得在该文件的开头处加上“storage.h”

头文件。

最后，不要忘记更改 Makefile 的 OBJS 变量，如下：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
interrupt.o mem.o driver.o ramdisk.o
```

好了，现在编译运行程序，运行结果如下：

```
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
Loaded RAM   ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

最终打印出来的值都是 0，恰好是“ram.img”的内容。读者也可以使用其他文件替换“ram.img”，来验证一下结果的正确性。

这样，一个最简单的存储设备驱动程序就完成了。当然在实际的系统应用中，我们必须还要考虑至少如下两个问题：第一个问题是，在极端情况下，操作系统可能同时会有七八个存储设备存在，那样的话，程序必须在初始化时，既调用 ramdisk_driver_init()又调用 nandflash_driver_init()，还要调用其他的 driver_init 函数。这样不但复杂，而且难以管理。因此，比较理想的方法是让驱动程序自行注册到系统中，而只在初始化的时候调用一次 all_driver_init()函数。另一个问题是存储设备分区的问题，通常的方法是定义结构体去描述每一个分区，而对这些分区的操作，则共享一组操作函数即可。

7.2 文件系统框架

现在，我们的操作系统已经具备了利用外部介质存储数据的条件，但这

并不意味着操作系统的数据管理部分就此完成了。

数据管理是操作系统的另一个重要核心。一个应用程序几乎不可能不需要对数据做处理，那么应用程序应该怎样去处理数据呢？这个问题乍看起来非常简单。在计算机的世界里，我们可以使用一组二进制数来描述所有的对象，数据自然也不例外。于是，在一个操作系统已经可以操作存储设备的前提下，将表示数据的一组二进制数顺序地写入存储器中，不就实现了数据的存储了吗？

没错，很多低端的操作系统的确是将数据直接存储在存储设备当中，但这样一点都不明智。

一方面，因为直接存储在存储设备上的数据很难管理，比如需要把个人的联系方式作为通讯录存储到 flash 中，如果数据不加处理直接存储，那么当我们想添加新的联系人、删除某个人的数据或者查找某个人的时候，就会变得非常困难。

另一方面，如果数据过于频繁地被删除和存储，就会很容易产生存储器碎片，前面章节中介绍的内存就是一个很典型的例子。当系统中碎片过多时，就会导致存储器利用率过低，并会降低数据读写的速度。还有一点是既然数据是直接存储在存储设备上的，这就要求用户至少在一定程度上了解存储设备的结构和原理，而对于一个存储设备众多的平台来说，就必须要求用户既要懂得 flash 的工作原理，又要懂硬盘、光盘的工作原理……很显然，这是非常不切实际的。

因此，高级操作系统都会想办法解决这个问题，不直接使用二进制数的形式去存储数据，而是将数据适当进行加工，使加工后的数据可以有效地克服上面几个缺点，然后把新的数据存储到设备当中。这种数据加工的格式和方法，我们就称它为文件系统。

7.2.1 文件系统的原理

文件系统其实是操作系统的一种抽象，它可以让存储设备的操作变得容易，用户不再需要了解存储器原理，而只需要知道文件系统的基本概念，如文件、文件夹等，就可以直接进行数据读写了。另外，由于要存储的数据是被重新加工后再被存储的，因此我们可以将数据的存储位置、大小等信息也

存到存储器中，这样，再进行数据查找、插入或删除时，就会变得相当容易了，就是因为这种抽象性，文件系统才不能代表一个具体的存储载体，而必须依托某一个具体的存储设备才会有意义。

我们刚才说实际要存储的数据在存储之前先要按照某种格式加工一下，再存储到存储器中，这种格式就叫做文件系统类型。读者一定会对 FAT 和 NTFS 这两种文件系统类型非常熟悉，这些都是在 Windows 系统下使用的默认文件系统类型。不同文件系统类型下的文件内容是一致的，只不过组织的方法不同罢了，而将某个存储设备组织成某种文件系统类型的过程就叫做格式化。文件系统在内核中的结构如图 7-1 所示。

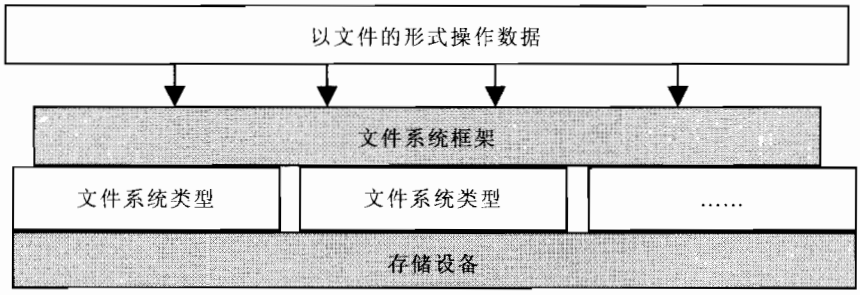

图 7-1　文件系统在内核中的层次结构

与驱动程序一致，本书中我们也不会将重点放在实现某种文件系统类型上，而是会从一个更高的层次出发去实现一个文件系统的框架。这样一来，任何一种文件系统都可以在这个框架下轻易地实现，从而从一个更高的角度去完成操作系统的设计。

现在，我们就正式进入文件系统框架的研究当中。在设计这个文件系统框架的时候，我们首先会想到的也许就是定义一个结构，用来描述文件系统上的一个对象。这里的对象就是我们平时所说的文件。

在很多操作系统中都用索引节点这个概念来描述这样一个结构，比如，在 minix 中，将索引节点称做 inode，在其他类 UNIX 操作系统中的命名方法也与此类似。索引节点列出了文件的属性等信息以及文件中各个数据块在磁盘中的相应位置。因此，我们可以得出如下结论：每个存储在存储器中的文件都会有一个索引节点与之对应。也可以说，索引节点就是存储在存储器中的数据的抽象。当我们想从设备中读取一段数据时，应该首先找到代表该

数据的索引节点，再调用与索引节点相配套的数据操作函数，通过索引节点中记录的数据信息正确地读出数据。这个函数在传统的 UNIX 操作系统中被称为 namei。使用 namei 能够得到对应文件的索引节点，而索引节点里又记录了必要的数据信息，于是我们就可以调用上一小节实现的存储设备驱动程序，将数据从索引节点中记录的实际数据存储位置中读出来，从而实现对数据的读取。

索引节点的意义如图 7-2 所示。

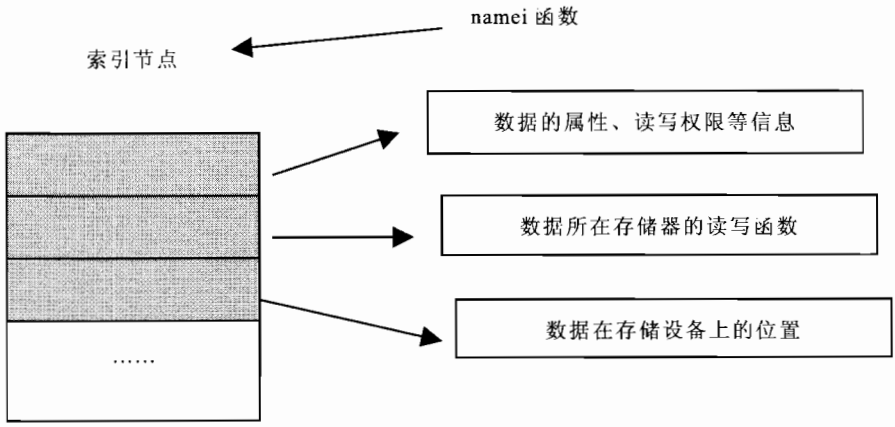

图 7-2 索引节点的意义

这样，我们的操作系统只需要根据某种文件系统类型的组织方式写出一个相应的 namei 函数，就可以支持该文件系统类型。当需要读出某个文件时，首先要知道这个文件是以哪种类型进行格式化的，还要知道数据被存储到了哪个存储器中，最后调用该文件系统类型的 namei 函数得到数据地址等信息，再调用相应的存储器读写函数进行实际的数据读写。

可以看出在实现了文件系统框架之后，无论文件系统类型怎样变化，是支持最新的文件系统类型还是剔除掉已经淘汰了的文件系统类型，都会变得很灵活。文件系统的框架是不变的，可变的只是与文件系统类型有关的代码。

7.2.2 文件系统框架的实现

好了，文件系统框架的原理我们已经搞清楚了。接下来让我们一起来实现一个最简单的文件系统框架吧！

代码 7-7

```
#include "storage.h"

#define MAX_SUPER_BLOCK    (8)
#define ROMFS  0

struct super_block;
struct inode{
    char *name;
    unsigned int flags;
    size_t dsize;
    unsigned int daddr;
    struct super_block *super;
};

struct super_block{
    struct inode *(*namei)(struct super_block *super,char *p);
    unsigned int (*get_daddr)(struct inode *);
    struct storage_device *device;
    char *name;
};

extern struct super_block *fs_type[];
```

代码 7-7 中包含了两个结构体，其中，struct inode 就是前文中提到的索引节点。

在该结构体中，成员 name 指向了该索引节点所代表的文件名，flags 成员用来实现对文件的控制。dsize 和 daddr 分别代表了文件的大小和它在存储器中的位置。从图 7-2 中索引节点的例子中可以看出，一个文件既包含了与文件系统相关的头信息又包含了真正的数据。在这里，成员 dsize 用来描述真实数据的大小，而 daddr 描述的则是整个文件在存储器中的实际位置，包含实际数据和相关的头信息。最后一个成员是一个结构体，名为 super_block，该结构体用来描述针对某个文件系统类型的控制数据和操作方法。每个文件系统类型都有一套这样的方法。这里，我们采用的都是与类 UNIX 操作系统中类似的命名方式。

struct super_block 结构体包含了前面提到过的 namei 函数指针，该指针指向的函数能够通过一个文件的文件名得到代表该文件的 inode 结构体。函数指针 get_addr 指向的函数能够计算头信息的大小，返回实际数据所在的位

置。成员 name 代表了该文件系统类型的名字，成员 device 是一个 storage_device 类型的结构体，它代表了这个文件系统类型所依存的存储设备。

代码 7-7 中还声明了一个 struct super_block 类型的数组，这个数组代表操作系统中所支持的文件系统列表。为了简化设计，我们采用数组的方式静态地管理每一个注册到该文件系统框架中的文件系统类型，与之相关的代码如下：

代码 7-8

```
#include "string.h"

#define MAX_SUPER_BLOCK 8
#define NULL (void *)0

struct super_block *fs_type[MAX_SUPER_BLOCK];

int register_file_system(struct super_block *type,unsigned int id){
    if(fs_type[id]==NULL){
        fs_type[id]=type;
        return 0;
    }
    return -1;
}

void unregister_file_system(struct super_block *type,unsigned int id){
    fs_type[id]=NULL;
}
```

代码 7-8 主要定义了两个函数，它们是 register_file_system 和 unregister_file_system。这两个函数的作用分别是将一个新的文件系统注册到文件系统框架中，以及从文件系统框架中删除一个已有的文件系统。这里的实现方法相当简单，无非就是操作 fs_type 这个数组的成员指向注册了的文件系统类型。系统还会给所支持的每一个文件系统类型都分配一个唯一的 ID，这个 ID 就代表了 fs_type 数组的索引，保证数组之间各成员不会冲突。

这样，一个基本文件系统框架就算完成了。与驱动程序框架类似，这段示例代码虽然功能简单，但是也足以说明文件系统框架的结构了。

接下来，我们就结合某一个文件系统类型来说明一下这样一个简单的文件系统框架是如何工作的。

7.2.3 romfs 文件系统类型

下面以某一个文件系统类型为例实现一个文件系统框架及其功能。

文件系统类型的种类非常多，除了前面提到过的 NTFS 等类型，比较知名的还包括 ext 系列、hfs 和 zfs，以及嵌入式环境下常常被提到的 jffs 系列和 cramfs，等等。甚至，我们自己也可以根据实际需要设计出自己的文件系统类型。

然而，综合比较各种文件系统类型的特点，本书最终决定以 romfs 为例来演示文件系统框架的作用，其原因主要包括如下两点。

第一，romfs 应该是功能完整的文件系统类型中最简单的一个了。romfs 文件系统广泛地被 Linux 以及 uClinux 所支持，其功能可谓完整，而且实现方法也相对简单，因为 romfs 是只读的文件系统，没有写的动作。在 Linux 中核心代码只有六七百行，相对于其他文件系统类型来说，这个尺寸简直太“苗条”了。因为我们的重点是文件系统框架，因此文件系统类型选择得越简单，就越能说明问题。

第二，romfs 在嵌入式领域里的应用也是非常广泛的。本书虽然展现了一个通用操作系统的完整结构，但重点仍然要侧重于嵌入式环境。因此可以说，romfs 文件系统类型非常合适。

既然选择了 romfs，那么在写代码之前，我们就必须要首先搞懂 romfs 的具体格式是怎样的，如图 7-3 所示。

首先，每一个 romfs 映像都包含一个文件系统头信息，用来记录整个文件系统类型的基本情况，比如，该 romfs 映像叫什么名字，一共有多少字节，等等。在 romfs 的文件系统头信息当中，前 8 个字节分别用来存储 '-'、'r'、'o'、'm'、'l'、'f'、's'、'-' 这 8 个字符，它们代表了 romfs 文件系统类型的标识符。

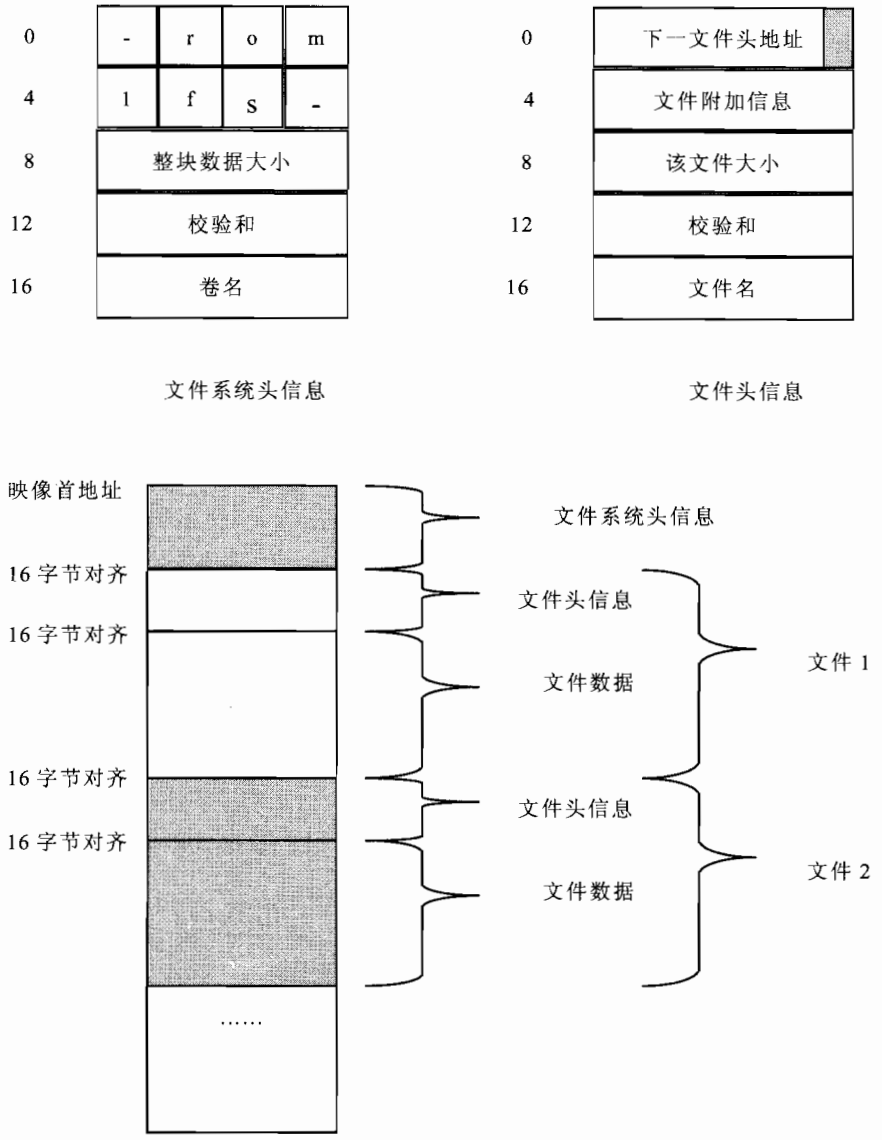

图 7-3 romfs 文件系统类型

在文件系统头信息结束的位置向下偏移 16 个字节处，就是该 romfs 映像中存储的第一个文件了。当然，首先出现的并不是数据本身，而是代表该段数据的文件头，其中记录了这个文件的名字、文件的数据部分有多大、文件是什么类型的文件等信息。这里，描述文件类型的信息由文件头信息最开始 4 个字节中的最后 3 个位 bit[0-2]来表示，也就是图中小方块的部分，而

bit[3]则代表了这个文件的执行属性。这些特点都是为了兼容 POSIX 标准而专门设计的。同时，文件头信息中的第 4~第 7 个字节说明了不同文件的附加信息，完整的说明如表 7-1 所示。

表 7-1　romfs 文件附加信息

数值（二进制）	含义	附加信息
000	硬链接	目的文件
001	目录	目录内首文件地址
010	普通文件	—
011	软链接	—
100	块设备文件	主、次设备号
101	字符设备	主、次设备号
110	socket 文件	—
111	fifo 文件	—

紧接着头信息后边的就是文件的实际数据了，当然这些数据的起始位置也需要按照 16 个字节向下对齐。

最后我们还要强调，romfs 中使用的所有数据都是大尾端的，这就要求在读取数据的时候，必须将其转换成 cpu 当前默认的端格式。

尾端这个概念我们前面从未提过，可能有些读者不清楚什么是端。通俗地讲，尾端就是在内存中，是用 0x01000000 来表示数字 1 还是用 0x00000001 来表示数字 1 的问题。

图 7-4 是大小尾端的数据存储示意图。

内存以字节为基本存储单位，那么当系统需要在内存中存储一个字节时，无论怎样都不会出现问题，但如果要存储的数据有两个字节呢？比如 0x0A0B 这个数，应该在低地址处存储 0x0A、高地址处存储 0x0B 还是相反呢？

很显然，存储的方式无所谓哪种正确，关键是要让所有数据都保持一致。这两种存储方式都是正确的，将低字节数存储在低端内存的位置上，就是小尾端格式，而将高字节数存储在低端内存上，就是大尾端格式。

读者也许紧接着就要问了，什么因素会影响尾端呢？

其实只要用心想一想就会知道，尾端跟 CPU 和人有关。在我们没有通知 CPU 采用什么尾端格式存储数据的前提下，CPU 会以本平台默认的方式完成数组存储。但对于某些平台来说，程序员也可以在编译程序时，人为地修改 CPU 的默认尾端方式。

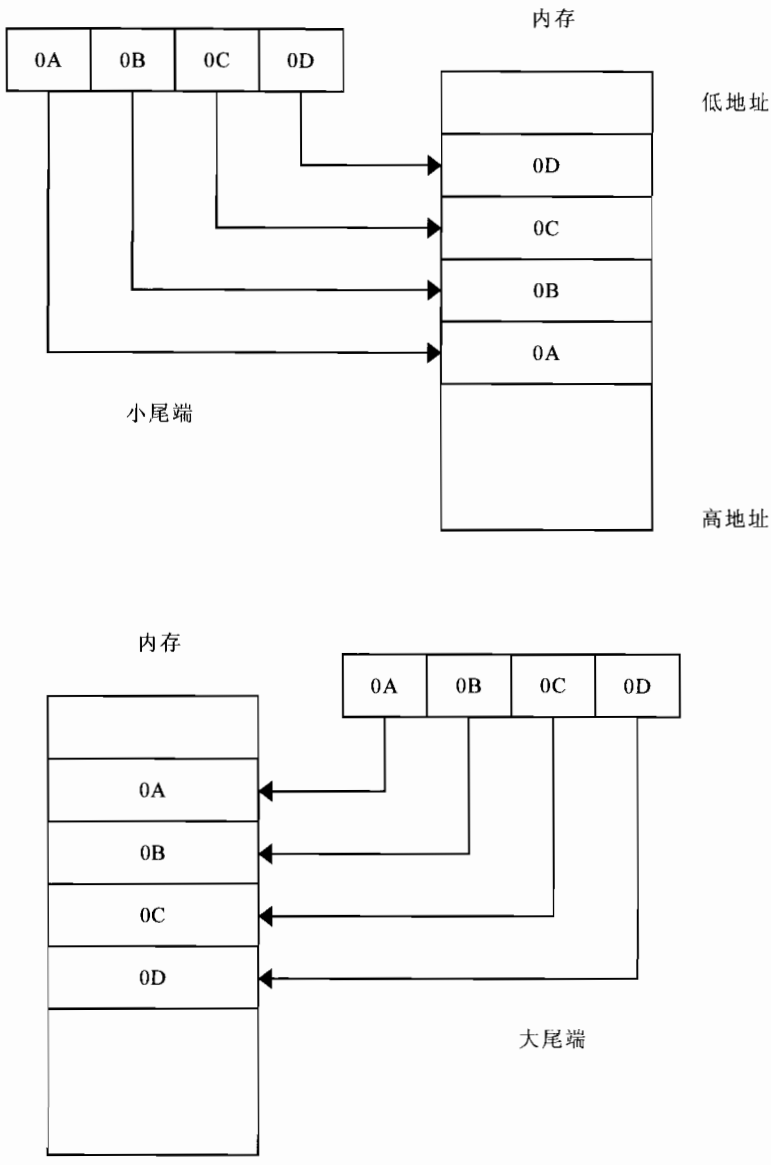

图 7-4 大小尾端示意图

现在一切都清楚了。ARM 体系结构在默认情况下采用小尾端格式存储数据，而我们也没有人为地更改这种默认格式，因此我们的操作系统使用的就是小尾端格式。

这样看来，将 romfs 文件系统类型从大尾端转换成小尾端的操作过程就

不可避免了。结合图 7-4 的描述，我们能够得出尾端转换的方法，只要将高位数据与低位数据互换即可。

7.2.4 实现 romfs 文件系统

此刻，在了解了 romfs 的存储格式后，我们就可以得出操作 romfs 文件系统类型的一般步骤了，代码的具体实现方法如下：

代码 7-9

```
#include "fs.h"
#include "storage.h"
#include "string.h"

#define NULL (void *)0

#define be32_to_le32(x) \
    ((unsigned int)( \
        (((unsigned int)(x) & (unsigned int)0x000000ffUL) << 24) | \
        (((unsigned int)(x) & (unsigned int)0x0000ff00UL) << 8) | \
        (((unsigned int)(x) & (unsigned int)0x00ff0000UL) >> 8) | \
        (((unsigned int)(x) & (unsigned int)0xff000000UL) >> 24) ))

struct romfs_super_block {
    unsigned int word0;
    unsigned int word1;
    unsigned int size;
    unsigned int checksum;
    char name[0];
};

struct romfs_inode {
    unsigned int next;
    unsigned int spec;
    unsigned int size;
    unsigned int checksum;
    char name[0];
};

struct super_block romfs_super_block;
```

```
#define     ROMFS_MAX_FILE_NAME       (128)
#define     ROMFS_NAME_ALIGN_SIZE (16)
#define     ROMFS_SUPER_UP_MARGIN (16)
#define     ROMFS_NAME_MASK\
              (~(ROMFS_NAME_ALIGN_SIZE-1))
#define     ROMFS_NEXT_MASK 0xfffffff0

#define romfs_get_first_file_header(p) \
              ((((strlen(((struct romfs_inode *)(p))->name)\
              +ROMFS_NAME_ALIGN_SIZE+\
              ROMFS_SUPER_UP_MARGIN)) &\
              ROMFS_NAME_MASK)<<24)

#define romfs_get_file_data_offset(p,num)  \
              (((((num)+ROMFS_NAME_ALIGN_SIZE)&\
              ROMFS_NAME_MASK)+\
              ROMFS_SUPER_UP_MARGIN+(p)))
```

代码 7-9 定义了实现 romfs 文件系统类型所需要的数据结构、变量和宏。

结构体 struct romfs_super_block 用于描述 romfs 文件系统头信息，而 struct romfs_inode 则用于描述 romfs 文件头信息。这两个结构体的作用和各成员的含义可以参考图 7-3 的描述。

宏 ROMFS_MAX_FILE_NAME、ROMFS_NAME_ALIGN_SIZE 等都是为了对 romfs 文件系统进行操作定义，是根据 romfs 文件系统类型格式而设计的。稍后我们可以看到这些宏在代码中的应用。

宏定义 romfs_get_first_file_header 专门负责找到整块磁盘中存储的第一个文件的位置。它的依据是 romfs 文件系统类型的 16 字节对齐的原则。如图 7-3 所示，只要知道了卷名的长度，再加上 struct romfs_super_block 结构体大小的偏移量，将这个值按照 16 字节对齐，就得到了第一个文件的位置了。

romfs_get_file_data_offset 宏可以通过文件头信息在存储器中的位置以及文件名的长度求出文件中实际数据所在位置，其工作方式与前面的宏类似。代码 7-9 中还有一个宏也是需要反复使用的，那就是 be32_to_le32，这个宏能将大尾端的数据转换成小尾端的数据。

介绍完了 romfs 文件系统类型相关的数据结构，我们再来研究一下文件系统框架中最重要的一个函数——namei 函数在 romfs 中的实现方法。

代码 7-10

```
static char *bmap(char *tmp,char *dir){
    unsigned int n;
    char *p=strchr(dir,'/');
    if(!p){
        strcpy(tmp,dir);
        return NULL;
    }
    n=p-dir;
    n=(n>ROMFS_MAX_FILE_NAME)?\
            ROMFS_MAX_FILE_NAME:n;
    strncpy(tmp,dir,n);
    return p+1;
}

static char *get_the_file_name(char *p,char *name){
    char *tmp=p;
    int index;
    for(index=0;*tmp;tmp++){
        if(*tmp=='/'){
            index=0;
            continue;
        }else{
            name[index]=*tmp;
            index++;
        }
    }
    name[index]='\0';
    return name;
}

struct inode *simple_romfs_namei(struct super_block *block,char *dir){
    struct inode *inode;
    struct romfs_inode *p;
    unsigned int tmp,next,num;
    char name[ROMFS_MAX_FILE_NAME],\
            fname[ROMFS_MAX_FILE_NAME];
    unsigned int max_p_size=(ROMFS_MAX_FILE_NAME+\
            sizeof(struct romfs_inode));
    max_p_size=max_p_size>(block->device->sector_size)\
            ?max_p_size:(block->device->sector_size);
    get_the_file_name(dir,fname);
```

```
    if((p=(struct romfs_inode *)kmalloc(max_p_size,0))==NULL){
        goto ERR_OUT_NULL;
    }
    dir=bmap(name,dir);
    if(block->device->dout(block->device,p,0,\
                block->device->sector_size))
        goto ERR_OUT_KMALLOC;
    next=romfs_get_first_file_header(p);

    while(1){
        tmp=(be32_to_le32(next))&ROMFS_NEXT_MASK;
        if(tmp>=block->device->storage_size)
            goto ERR_OUT_KMALLOC;
        if(tmp!=0){
            if(block->device->dout(block->device,p,\
                    tmp,block->device->sector_size)){
                goto ERR_OUT_KMALLOC;
            }
            if(!strcmp(p->name,name)){
                if(!strcmp(name,fname)){
                    goto FOUND;
                }else{
                    dir=bmap(name,dir);
                    next=p->spec;
                    if(dir==NULL){
                        goto FOUNDDIR;
                    }
                }
            }else{
                next=p->next;
            }
        }else{
            goto ERR_OUT_KMALLOC;
        }
    }

FOUNDDIR:
    while(1){
        tmp=(be32_to_le32(next))&ROMFS_NEXT_MASK;
        if(tmp!=0){
if(block->device->dout(block->device,p,tmp,\
                        block->device->sector_size)){
                goto ERR_OUT_KMALLOC;
```

```
            }
            if ( !strcmp ( p->name,name )) {
                goto FOUND;
            }else{
                next=p->next;
            }
        }else{
            goto ERR_OUT_KMALLOC;
        }
    }

FOUND:
    if ((inode = (struct inode *) kmalloc (sizeof (struct inode), 0)) \
                                        ==NULL ) {
        goto ERR_OUT_KMALLOC;
    }
    num=strlen ( p->name ) ;
    if (( inode->name= ( char * ) kmalloc ( num,0 )) ==NULL ) {
        goto ERR_OUT_KMEM_CACHE_ALLOC;
    }
    strcpy ( inode->name,p->name ) ;
    inode->dsize=be32_to_le32 ( p->size ) ;
    inode->daddr=tmp;
    inode->super=&romfs_super_block;
    kfree ( p ) ;
    return inode;

ERR_OUT_KMEM_CACHE_ALLOC:
    kfree ( inode ) ;
ERR_OUT_KMALLOC:
    kfree ( p ) ;
ERR_OUT_NULL:
    return NULL;
}
```

simple_romfs_namei这个函数就是romfs中namei函数的具体实现方法。这个函数比较复杂，我们对它的介绍会相对宏观一些。

首先，想要让simple_romfs_namei函数能够正常工作，就得依赖两个辅助函数。其中一个函数是get_the_file_name，它能够除去文件的路径，得到文件名，另一个函数是bmap，函数bmap每次运行的时候，都可以将文件名字符串中最顶层的目录名去掉，然后返回余下的目录名。例如，一个文件

的完整路径名保存到下面这个变量中：

```
char *name="firstdir/seconddir/filename"
```

那么，当我们调用 bmap 函数时：

```
char *newname=bmap(buf,name);
```

函数返回值 newname 就应该是 seconddir/filename 字符串，而 buf 内存区存储的字符串就应该是 firstdir 了。simple_romfs_namei 函数正是反复调用了 bmap 函数，一次次剥掉路径名后，找到了文件的具体位置。

在 simple_romfs_namei 函数的一开始，程序首先定义了一些变量，获取最终文件名、分配内存空间。其中，临时变量 p 代表新申请的内存空间的首地址。通过 get_the_file_name 函数，数组 fname 保存了不含路径的文件名。字符数组 name 专门给 bmap 函数使用，保存每次得到的文件路径。

紧接着，程序通过 super_block 中 device 成员的 dout 成员函数指针读取存储器中最开始的一块数据。成功读出数据之后，调用代码 7-9 中的 romfs_get_first_file_header 宏，就可以得到第一个文件在存储器中的位置了。

然后，通过两个大循环，结合 bmap 函数，一层层地查找文件的各级路径。在第一个循环中，临时变量 tmp 保存了下一个要读取的文件在存储器中的位置，根据 romfs 文件系统的规则，每一个文件头信息的 next 成员都指向了下一个文件的具体位置。程序读取这个值，然后通过 be32_to_le32 转换成小尾端，赋值给 tmp 变量。之后再次调用 dout 函数，读出新的文件内容并判断这个新的文件是不是我们要查找的文件，或者至少是我们要查找的文件的某个父目录，如果没有找到，则进行下次循环，如果找到的是文件的所属目录，则跳转到 FOUNDDIR 标签处，查找最终的文件。

如果文件最终被找到，则跳转到 FOUND 标签，在此处，程序首先分配一个 struct inode 空间，然后根据找到文件的相关头信息填充 struct inode 的相应成员。最后释放掉不使用的空间，将 struct inode 指针返回。

函数 simple_romfs_namei 成功得到了文件系统框架下的 struct inode 结构体。此时，我们再通过另外一个函数就可以使 romfs 这一文件系统类型在框架下发挥作用了。

代码 7-11

```
unsigned int romfs_get_daddr(struct inode *node){
```

```
    int name_size=strlen(node->name);
    return romfs_get_file_data_offset(node->daddr,name_size);
}

struct super_block romfs_super_block={
    .namei=simple_romfs_namei,
    .get_daddr=romfs_get_daddr,
    .name="romfs",
};

int romfs_init(void){
    int ret;
    ret=register_file_system(&romfs_super_block,ROMFS);
    romfs_super_block.device=storage[RAMDISK];
    return ret;
}
```

这个函数就是romfs_get_daddr，它能够从一个标准的struct inode结构体中得到文件实际数据的存储位置。这个函数的实现非常简单，只需要计算文件名所占的字节数，然后调用romfs_get_file_data_offset宏就可以了，这个宏被定义在了代码7-9中。

现在，我们需要定义一个文件系统框架下的super_block结构体为romfs_super_block，然后将其中的namei和get_daddr成员分别赋值。另外，我们还需要有一个针对romfs的初始化函数romfs_init，在该函数中，首先需要将romfs_super_block的device成员赋值，因为文件系统类型是一个抽象的格式，而它最终必须寄生在某个具体的存储设备中，才能发挥作用。因此，我们必须在romfs初始化的时候，就为它选定哪个设备使用了该文件系统类型去管理文件。在Linux中，这一工作是通过mount命令完成的，在这里我们只是简单地将这个信息直接写到代码当中去。

最后，romfs_init函数还需要调用框架中的register_file_system函数，将romfs_super_block注册到系统中去。

这样，基于romfs的文件系统类型就可以在我们的操作系统中发挥作用了。

7.2.5 让代码运行起来

读者朋友们一定迫不及待地想要看一下，怎样在romfs文件系统类型中

读取文件吧？接下来我们就借助 RAM 盘，在其中存储一些实际的文件，然后利用文件系统框架下的方法来读取这些文件的内容。

首先，我们要将前面的代码添加到我们自己的操作系统中进行编译。

请将代码 7-7 的内容保存成名为“fs.h”的文件，将代码 7-8 命名为“fs.c”，这两个都是与文件系统框架有关的。然后，将代码 7-9、代码 7-10 和代码 7-11 的内容合并成一个文件，取名为“romfs.c”。所有与 romfs 文件系统类型有关的代码都保存在该文件中。

此时，我们还需要另外一个文件来提供一些必要的函数。因为在我们写操作系统的一开始，就没有使用标准库函数，所以，前面程序中使用的诸如 memcpy 之类的函数就必须由我们自己实现了。好在实现这些函数并不是什么难事，代码 7-12 恰好能完成这些工作。

代码 7-12

```
#ifndef __STRING_H__
#define __STRING_H__

static inline int strcmp(const char * cs,const char * ct){
    register signed char __res;
    while (1) {
        if ((__res = *cs - *ct++) != 0 || !*cs++)
            break;
    }
    return __res;
}

static inline void * memset(void * s,int c,unsigned int count)
{
    char *xs = (char *) s;
    while (count--)
        *xs++ = c;
    return s;
}

static inline char * strcpy(char * dest,const char *src){
    char *tmp = dest;
    while ((*dest++ = *src++) != '\0');
    return tmp;
}
```

```
static inline char * strncpy(char * dest,const char *src,\
unsigned int count){
    char *tmp = dest;
    while (count-- && (*dest++ = *src++) != '\0');
    return tmp;
}

static inline unsigned int strlen(const char * s){
    const char *sc;
    for (sc = s; *sc != '\0'; ++sc);
    return sc - s;
}

static inline char * strchr(const char * s, int c){
    for(; *s != (char) c; ++s)
        if (*s == '\0')
            return (void *)0;
    return (char *) s;
}

#endif
```

这些函数的具体实现我们就不多解释了，相信读者一定看得明白。这里我们使用了一个小技巧，那就是将所有的函数都定义成 inline。inline 函数代表着凡是使用 inline 函数的地方都要将函数展开，而不是去调用这个函数，同时在编译的时候，也对函数参数进行了类型检查。所以，使用 inline 函数要比使用宏定义稍安全一些，而它的缺点就是会增大最后生成的代码的尺寸。

普通的函数调用只需要知道函数的入口地址就可以了，因此在将程序编译成目标文件时，并不需要知道函数是怎样实现的。而 inline 函数则不同，因为需要原地展开，所以程序在使用 inline 函数之前，必须知道函数是怎样实现的。因此，一个 inline 函数通常被定义在头文件中，然后让别的程序包含这个头文件。但这样会带来另外一个问题，当两段程序包含了同一个头文件时，编译时就会出现同一个函数被定义两次的情况，这在 C 语言中是不允许的。所以在 inline 函数定义时，还要加上 static 关键字，表示该函数是私有的。

我们将代码 7-12 保存成文件，命名为“string.h”。然后修改 Makefile

中的 OBJS 变量为如下形式：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
interrupt.o mem.o driver.o ramdisk.o romfs.o fs.o
```

此时，文件系统框架和 romfs 文件系统类型的代码就基本添加完成了。在运行这段代码之前，我们还需要做一个额外的工作。不要忘记，现在系统中的“ram.img”文件镜像中并没有被格式化成 romfs 的文件系统类型，这个过程需要一些工具的辅助。

为了使代码结构更清晰，我们需要新建一个目录，将格式化 romfs 的工具保存到这个目录当中。无论在 Linux 系统中还是 Cygwin 模拟环境下，都可以使用 mkdir 命令。于是，请先使用 cd 命令切换到代码的根目录下，我们需要在程序根目录下运行如下命令：

命令 7-1

```
mkdir tools
```

然后，进入该文件夹中。

命令 7-2

```
cd tools
```

读者需要从 romfs 的网站或者网站 www.leeos.org 上下载一个名为“genromfs.c”的源程序到当前文件夹，使用下面这条命令，就可以生成一个能够格式化 romfs 文件系统类型的工具了。

命令 7-3

```
gcc genromfs.c
```

这样会生成一个名为“a.out”的文件。

同时，我们需要在 RAM 盘中存储一个文件。请在 tools 目录中新建另一个文件夹，取名为“filesystem”。

命令 7-4

```
mkdir filesystem
```

文件夹“filesystem”里的内容，就代表了存储在 RAM 盘根目录中的内容。于是我们可以使用 echo 命令，在 RAM 盘中生成一个文件，里面存储 0~9 共 10 个数字。

命令 7-5

```
echo "0123456789" > filesystem/number.txt
```

然后，使用刚才编译成的 a.out 命令生成一个 romfs 文件系统格式的文件镜像，取名为“romfs.img”。

命令 7-6

```
./a.out -d filesystem/ -f romfs.img
```

最后，使用 dd 命令将“romfs.img”的内容写到我们之前使用过的“ram.img”文件当中。如果读者就是按照本书的方法进行操作的话，这个文件就应该在父目录中。于是，运行如下命令：

命令 7-7

```
dd if=./romfs.img of=../ram.img
```

这样会让“ram.img”文件变小，不过这不会影响到程序的正常运行。

好了，现在是时候使用程序读取 RAM 盘中的文件了。我们需要修改“boot.c”文件。

代码 7-13

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
//  timer_init();

    init_page_map();
    kmalloc_init();
    ramdisk_driver_init();
    romfs_init();

    struct inode *node;
    char buf[128];

    node=fs_type[ROMFS]->namei(fs_type[ROMFS],"number.txt");
    fs_type[ROMFS]->device->dout(fs_type[ROMFS]->device,\
```

```
            buf,fs_type[ROMFS]->get_daddr(node),\
            node->dsize);

    for(i=0;i<sizeof(buf);i++){
        printk("%c ",buf[i]);
    }

    while(1);
}
```

在代码 7-13 中，程序首先使用了 romfs_init 函数进行 romfs 文件系统的初始化。然后调用 fs_type[ROMFS]的 namei 函数，从 RAM 盘中读取“number.txt”文件，得到代表该文件的 inode 结构体。接下来，程序调用 device 成员的 dout 函数，从该结构体中读取实际数据到 buf 中。最后，通过循环将 buf 中的值以字符的形式打印出来，不出意外，打印的结果就应该是 0~9。

最后，不要忘记在“boot.c”文件的最开始处添加“fs.h”头文件。

一切都准备好后，就可以编译运行程序，运行的结果如下：

```
Your elf file is little endian.
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x426c70
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
Loaded RAM   ./ram.img
start addr is set to 0x30000000 by exec file.
helloworld
1 2 3 4 5 6 7 8 9 0
```

7.3 总结

相信读者已经理解了框架的真正作用和意义。

从程序员的角度来看，我们必须承认一个现实，那就是——需求的变化

是项目过程中唯一不变的东西。在日常的工作过程中，相信很多读者都会对这句话深有感触。

如果读者朋友们能够承认这一点，就不会去不遗余力地追求那些不可能实现的完美程序，或者要求别人一开始就提出确定不变的完整需求，而是会努力地让自己的程序尽最大可能适应一切变化，能够在变化中投入的资源最少却收益最大。想要实现这一点，就需要把握一条最基本的设计原则，那就是将相对不变的对象独立出来，而让勤于变化的对象去随意改变，同时尽可能地降低二者之间的耦合。

框架正是因此而诞生的。有了框架，就拥有了一套相对稳定的结构，此时，它就成为了相对不变的对象，在需求的变化过程中，框架就会相对静止，而那些依存框架的部分也就相对静止了。将那些依托于框架的实现安排在框架之外，让这一部分因时而动，并保持二者之间的低耦合。

这样一来，我们便拥有了主动应对需求变化的致命武器。

在我们的操作系统中，文件系统和驱动程序框架都是为此而设计的。虽然代码短小、功能单一，但其背后的思想却是强大的。因为框架结构设计的相对不变，所以依存于框架之上的应用程序就可以保持相对稳定，当我们需要在操作系统中实现新的文件系统或驱动程序时，只需要实现框架之下的接口，而框架之上的一切程序都无须改变。

用这样的设计思路和方法来写程序，我们的程序就能够实现真正的“永生”。